QFilter: Towards a Fine-Grained Access Control for Aggregation Query Processing over Secret Shared Data

Meghdad Mirabi[1,*], Carsten Binnig[1]

[1]Faculty of Computer Science, Technical University of Darmstadt, Darmstadt, Germany

Abstract

This paper presents QFilter, a privacy-preserving and communication efficient solution that integrates an Attribute-Based Access Control (ABAC) model into query processing. QFilter enables the specification and enforcement of fine-grained access control policies tailored to secret-shared data. It can process aggregation SQL queries, including "count", "sum", and "avg" functions, with both conjunctive (using "AND") and disjunctive (using "OR") equality query conditions, without the need for inter-server communication. QFilter is secure against *honest-but-curious* adversaries, and the preliminary experiments illustrate its applicability for preserving privacy in query processing over secret-shared data, especially at the tuple level access control with the lowest overhead.

Keywords

Access Control, Data Outsourcing, Privacy Preserving, Query Processing, Secret Sharing

1. Introduction

Cloud computing offers numerous benefits to organizations and individuals looking to store and process data in a public environment. These advantages include high availability, scalability, and efficiency, while also reducing infrastructure provisioning and maintenance expenses [1, 2, 3]. However, security concerns pose significant obstacles to data outsourcing in the cloud. There is apprehension about potential breaches of data privacy and leakage of data processing results to other cloud tenants or the service provider, as data owners lack direct control over their data and computations [4, 5, 6].

To enhance the security of outsourced data, it is essential to develop models and mechanisms for specifying and enforcing access control policies tailored to this context [7, 8]. These models should consider varying levels of data sensitivity. Fine-grained access control policies offer enhanced control, allowing organizations to define restrictions at the level of individual tuples, attributes, or cells within a relation [9, 10, 11]. Such granular control ensures the protection of sensitive data by permitting access only to authorized users. In addition to the access control specification model, it is crucial to tightly integrate an access control enforcement mechanism into the query processing workflow for data retrieval [12, 13, 14, 15]. By seamlessly integrating access control enforcement into the query processing engine, unauthorized access attempts can be promptly thwarted, while authorized users can efficiently perform their operations. The enforcement mechanism must prioritize privacy, safeguarding sensitive data when processed by external servers. Furthermore, it should address efficiency and scalability challenges, considering the substantial volumes of data involved in query processing over outsourced data.

The conventional approach to protect sensitive data from unauthorized disclosure is to encrypt the data before outsourcing it [5, 16, 17, 11, 18, 19]. This approach involves transferring access to the outsourced data into access to secret keys used for encryption prior to uploading it to the cloud. However, existing techniques such as Homomorphic Encryption [20, 21, 22, 23, 24], Searchable-Encryption [25, 26, 27, 28], and Bucketization [29] are either extremely complex or cannot practically support various types of queries.

Attribute-Based Encryption (ABE) is an effective technique in cloud computing for ensuring data confidentiality, data privacy, and fine-grained access control [30, 31, 32, 33, 34, 35, 36, 37]. ABE allows decryption of ciphertext only if the user's attribute set meets the specified access control policies. However, conventional ABE approaches suffer from the drawback of revealing user attributes and access policies to the public, making them susceptible to inference attacks [38, 39].

Recently, several research works have explored secret sharing schemes for efficient processing of aggregation SQL queries over outsourced data while preserving the

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23) — Workshop on Cloud Databases (CloudDB'23), August 28 - September 1, 2023, Vancouver, Canada

*Corresponding author.

meghdad.mirabi@cs.tu-darmstadt.de (M. Mirabi); carsten.binnig@cs.tu-darmstadt.de (C. Binnig)

https://www.informatik.tu-darmstadt.de/systems/systems_tuda/group/team_detail_120640.en.jsp (M. Mirabi); https://www.informatik.tu-darmstadt.de/systems/systems_tuda/group/team_detail_18624.en.jsp (C. Binnig)

0000-0003-3803-2756 (M. Mirabi); 0000-0002-2744-7836 (C. Binnig)

privacy of the data, user queries, and query results. However, the existing proposals [40, 41, 42, 43] assume that data users are fully trusted and can access outsourced data without any limitations. In practice, data users have different access privileges, and requests to access unauthorized parts of outsourced data must be filtered.

Existing research on data outsourcing lacks a practical solution for specifying fine-grained access control policies and enforcing them during the processing of aggregation SQL queries over secret-shared data while simultaneously preserving the privacy of outsourced data, associated access control policies, user queries, and query results. To address these challenges, this paper introduces QFilter, which integrates an Attribute-Based Access Control (ABAC) model with aggregation query processing. QFilter enables the specification and enforcement of fine-grained access control policies tailored to secret-shared relations. The proposed ABAC model supports flexible access control policy specification at the tuple, attribute, or cell level, accommodating complex access conditions. To enforce access authorizations, QFilter employs an oblivious query rewriting and processing technique, incorporating query conditions into the WHERE clause of the submitted SQL query to check access authorizations and filter unauthorized data during query execution. For efficient and privacy-preserving query processing, QFilter utilizes string matching-based operators to process aggregation SQL queries, including "count", "sum", and "avg" functions, with both conjunctive (using "AND") and disjunctive (using "OR") equality query conditions over secret-shared data. The contributions of this paper can be summarized as follows:

- We propose an ABAC model for QFilter that introduces new attributes to the relation, representing specific access control policies. These attributes allow us to specify a set of fine-grained access control policies. To support complex access conditions and reduce the number of ABAC policies, our proposed ABAC model combines user attribute conditions using boolean operators, resulting in a single condition set. This set is then automatically mapped to a unique user group in QFilter, which facilitates the specification of access control policies for different data items (i.e., tuples, attributes, or cells) in the outsourced relation.
- We obliviously rewrite the submitted aggregation SQL query in QFilter by adding new query conditions to the WHERE clause, ensuring access authorizations are checked and unauthorized data items are filtered out during query processing. Importantly, QFilter eliminates the need for inter-server communication during query rewriting.
- We design efficient string matching-based operators for QFilter to obliviously process aggregation SQL queries, including "count", "sum", and "avg" functions, with both conjunctive (using "AND") and disjunctive (using "OR") equality query conditions over secret-shared data. These operators utilize bit-wise multiplication and addition on secret-shared data, enabling fast computation of aggregation results. Moreover, the use of these operators in QFilter eliminates the need for inter-server communication during query processing.
- We analyze QFilter in terms of the number of computation and communication rounds at both server and data user sides. Additionally, we conduct preliminary experiments to demonstrate the performance overhead of QFilter for data outsourcing and privacy-preserving query processing.

The rest of paper is organized as follows: Section 2 describes the preliminary concepts. Section 3 provides an overview of the system architecture, adversary model, and security requirements in QFilter. Section 4 presents our proposed ABAC model. Section 5 explains our proposed solution for data outsourcing and oblivious query rewriting and processing. Section 6 evaluates the overhead of QFilter for data outsourcing and privacy preserving query processing. Section 7 reviews and compares existing research works with QFilter. Finally, Section 8 concludes the paper and discusses future works.

2. Background

In this section, we briefly review the basic concepts used in QFilter.

2.1. Shamir's Secret Sharing Scheme

Shamir's secret sharing scheme [44] is a threshold secret sharing scheme that provides security against adversaries with unlimited computing resources. The basic idea behind Shamir's secret sharing scheme is that k points are enough to define a k-1 degree polynomial. To share a secret value S among c non-communicating participants/servers, the data owner chooses k-1 random coefficients a_1, a_2, ..., a_{k-1} to build a polynomial $f(x) = a_0 + a_1x^1 + a_2x^2 + ... + a_{k-1}x^{k-1}$, where ($k \leq c$), $f(x) \in \mathbb{F}_{\mathbb{P}}[x]$, $\mathbb{P}$ is a prime number, $\mathbb{F}_{\mathbb{P}}$ is a finite field of order $\mathbb{P}$, $a_0 = S$, and $a_i \in \mathbb{N}$ ($\forall 1 \leq i \leq k-1$). Then, each participant/server i ($\forall 1 \leq i \leq c$) is given a point $(x_i, f(x_i))$ on the polynomial. The secret value S can be reconstructed by performing the Lagrange interpolation operation using any subset of k secret shares [45, 46, 47].

2.2. String Matching on Secret Shares

Recently, a new string matching method called Accumulating Automata (AA) is proposed by [48], which eliminates the requirement for cooperation between participants/servers during string matching. This technique can be effectively utilized in QFilter to determine the satisfaction of query conditions in the WHERE clause of submitted SQL queries.

Assuming that S is the secret value and S_i ($\forall 1 \leq i \leq c$) represents the ith secret-share of S stored at the corresponding server, the AA method enables a data user to search a string pattern p. By generating c secret-shares of p (p_i, $\forall 1 \leq i \leq c$), each server can independently search for the secret-share pattern p_i within the secret-share S_i. The result is a secret-share of either 1 or 0, indicating a match or mismatch between the respective secret-shares. The AA method combines the secret-shares through multiplication and addition, allowing the data user to reconstruct the secret value using the Lagrange interpolation operation after collecting outputs from k servers, where $k \leq c$. The process of string matching using this method in shown in Example 1.

Example 1. *Consider the values of the Account Type attribute in the Account relation shown in Table 1, which are "checking" and "saving". These two values can be mapped to "01" and "10" in the unary representation form, respectively, as we only have these two values for the Account Type attribute.*

Table 1
Account Relation

Account No.	Account Type	Balance (×1000$)
1	checking	2
2	saving	3
3	checking	1

Now, let's assume that the unary representation "01" of the value "checking" is outsourced by the data owner. This means that the value "checking" will be revealed to the adversary. To prevent data disclosure, the data owner can use two polynomials with an identical degree to outsource the value "checking" to a set of 5 non-communicating servers, as shown in Table 2.

Table 2
Secret-Shares of Value "checking" Created by the Data Owner

Value	Polynomial	S1 (x=1)	S2 (x=2)	S3 (x=3)	S4 (x=4)	S5 (x=5)
0	0+x	1	2	3	4	5
1	1+2x	3	5	7	9	11

Now, assume that the data user wants to search for the value "checking". The data user knows that the value "checking" is represented as "01". Then, he/she creates secret-shares for that as shown in Table 3. It should be noted here that the data user does not need to ask from the data owner about any polynomial to build the secret-shares of value "checking".

Table 3
Secret-Shares of Value "checking" Created by the Data User

Value	Polynomial	S1 (x=1)	S2 (x=2)	S3 (x=3)	S4 (x=4)	S5 (x=5)
0	0+3x	3	6	9	12	15
1	1+4x	5	9	13	17	21

At the server side, every individual server performs a position-wise multiplication of the bits they possess, adds up all the multiplication results, and sends them to the data owner. This process is illustrated in Table 4.

Table 4
Operations Performed by Non-Communicating Servers

Server 1	Server 2	Server 3	Server 4	Server 5
$1 \times 3 = 3$	$2 \times 6 = 12$	$3 \times 9 = 27$	$4 \times 12 = 48$	$5 \times 15 = 75$
$3 \times 5 = 15$	$5 \times 9 = 45$	$7 \times 13 = 91$	$9 \times 17 = 153$	$11 \times 21 = 231$
$3 + 15 = 18$	$12 + 45 = 57$	$27 + 91 = 118$	$48 + 153 = 201$	$75 + 231 = 306$

After receiving the outputs from the set of 5 non-communicating servers, which are y_1 = 18, y_2 = 57, y_3 = 118, y_4 = 201, and y_5 = 306, the data user performs the Lagrange interpolation operation to reconstruct the secret answer, which is 1 (i.e., b_0 = 1), confirming that the string pattern has been found. The process is as follows:

$$\frac{(x-x_2)(x-x_3)(x-x_4)(x-x_5)}{(x_1-x_2)(x_1-x_3)(x_1-x_4)(x_1-x_5)} \times y_1 + \frac{(x-x_1)(x-x_3)(x-x_4)(x-x_5)}{(x_2-x_1)(x_2-x_3)(x_2-x_4)(x_2-x_5)} \times y_2 +$$

$$\frac{(x-x_1)(x-x_2)(x-x_4)(x-x_5)}{(x_3-x_1)(x_3-x_2)(x_3-x_4)(x_3-x_5)} \times y_3 + \frac{(x-x_1)(x-x_2)(x-x_3)(x-x_5)}{(x_4-x_1)(x_4-x_2)(x_4-x_3)(x_4-x_5)} \times y_4 +$$

$$\frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)}{(x_5-x_1)(x_5-x_2)(x_5-x_3)(x_5-x_4)} \times y_5 = \frac{(x-2)(x-3)(x-4)(x-5)}{(1-2)(1-3)(1-4)(1-5)} \times 18 +$$

$$\frac{(x-1)(x-3)(x-4)(x-5)}{(2-1)(2-3)(2-4)(2-5)} \times 57 + \frac{(x-1)(x-2)(x-4)(x-5)}{(3-1)(3-2)(3-4)(3-5)} \times 118 +$$

$$\frac{(x-1)(x-2)(x-3)(x-5)}{(4-1)(4-2)(4-3)(4-5)} \times 201 + \frac{(x-1)(x-2)(x-3)(x-4)}{(5-1)(5-2)(5-3)(5-4)} \times 306 =$$

$$b_0 + b_1x^1 + b_2x^2 + b_3x^3 + b_4x^4, b_0 = 1$$

3. Overview

In this section, we provide an overview of the system architecture of QFilter, the adversarial model, and the security requirements in QFilter.

3.1. System Architecture

We assume three entities in QFilter's system architecture: *Data Owner*, *Data User*, and *Non-Communicating Servers*. The interaction between these entities are shown in Figure 1.

In Step 1, the data user registers his identity attributes with the data owner. In Step 2, the data owner creates and sends a credential to the data user based on his identity attributes. User credentials are stored in the *userInfo*

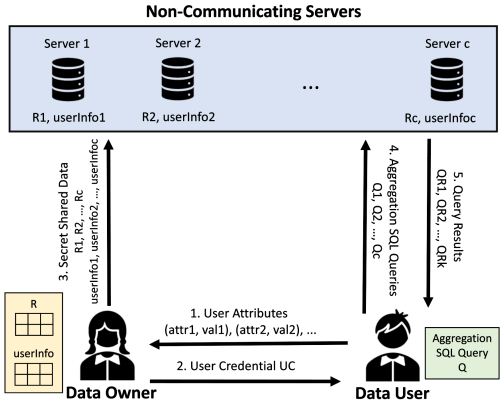


Figure 1: System Architecture of QFilter

relation at the data owner side, and authentication is performed using these credentials at the server side. In Step 3, the data owner splits relation R (i.e., including data and associated access control policies) into c relations R_1, R_2, ..., and R_c using Shamir's secret sharing scheme, and sends each relation to the corresponding server. Similarly, c relations $userInfo_1$, $userInfo_2$, ..., and $userInfo_c$ are created for relation *userInfo* and sent to the respective servers. Every individual server S_i requires the information stored in its corresponding $userInfo_i$ relation for the query rewriting process. In Step 4, the data user creates c queries Q_1, Q_2, ..., and Q_c based on the query Q by converting each value of query condition into sets of secret-shared values. Each query is sent to the corresponding server along with the set of secret-shared values. In Step 5, every individual server rewrites its assigned query by adding new query conditions to the WHERE clause for access authorization checks and executes it to produce the partial output for that. The partial outputs from the execution of queries Q_1, Q_2, ..., and Q_k (where $k < c$) are then sent to the data user. The data user then performs the Lagrange interpolation operation to obtain the final query result.

3.2. Adversary Model

In this paper, we consider the *honest-but-curious* adversary model. Under this model, non-communicating servers faithfully perform their assigned tasks (i.e., query rewriting and query processing) without attempting to modify sensitive data. However, every individual server may utilize side information, such as background knowledge, query execution, and output size, to gather useful information about outsourced data, access control policies, user query, and query results.

We assume that the data owner is fully trusted and immune to any attacks from the adversary. The adversary also lacks access to the secret-sharing algorithm and its related information employed by the data owner.

In this model, only authenticated users are allowed to request aggregation SQL queries on the outsourced data. They seek to uncover unauthorized portions of the data and confidential access control policies. Additionally, they aim to extract information about filtered parts of their query results. Authenticated users establish secure and trusted communication channels with the data owner and non-communicating servers, employing their credentials for authentication at the server side.

It is important to note that QFilter adheres to the limitations imposed by secret sharing schemes, where the adversary is unable to collude with the majority of servers or access the communication channel between the data owner/data user and individual servers. It is assumed that the servers are hosted across diverse cloud infrastructures, each managed by a distinct cloud service provider. Additionally, it is assumed that the majority of these providers refrain from colluding with one another due to their conflicting interests.

3.3. Security Requirements

In the *honest-but-curious* adversary model, QFilter must prevent an adversary from learning about the outsourced data itself, associated access control policies, user queries, and query results. This prevention is essential to avoid potential attacks, including:

- *Frequent Count Attack*: Observing cryptographically secure data and associated access control policies to infer the frequency of each value.
- *Access Pattern Attack*: Deducting which tuples satisfy or are filtered out from a submitted query.
- *Query Pattern Attack*: Analyzing the pattern of queries issued by data users to infer sensitive information or unauthorized access patterns.
- *Output Size Attack*: Counting the number of tuples satisfying or being filtered out from a query condition during both query processing and access authorization checking.

The privacy of secret values (i.e., outsourced data itself, associated access control policies, and query results) in QFilter relies on two factors: (1) ensuring that only the data owner or authorized users can reconstruct the secret value, and (2) providing unique representations for each occurrence of a value at each individual server to prevent frequency analysis.

To ensure the privacy of user queries in QFilter, two conditions must be met: (1) the actual values of query conditions are concealed from adversaries, and (2) queries of the same type cannot be distinguished based on their query results. Queries are considered to be of the same

type if they produce identical output sizes, such as aggregation SQL queries containing the "count" function.

QFilter must ensure that every individual server behaves identically when processing a given aggregation SQL query submitted by the data user. Furthermore, the servers must always provide the same answer to the query. To demonstrate this, it is needed to compare the actual execution of the algorithms used in QFilter for query processing on the servers with the ideal execution of these algorithms at a trusted party that has the same data, access control policies, and query conditions. An algorithm in QFilter preserves data privacy from every individual server if the real and ideal executions of such an algorithm return the same answer to the data user.

4. Access Control Specification

In this section, we provide a detailed description of our proposed access control model.

4.1. Access Control Moded in QFilter

In Attribute Based Access Control (ABAC) model, request of a subject to perform an action on an object is granted or denied based on a set of the assigned attributes of subjects and objects [49, 50, 51, 52]. Based on the general basis in this model, our proposed ABAC model includes the following elements:

- ***Users* (*U*)**. It consists of the data users who submit aggregation SQL queries over the outsourced data in QFilter.
- ***Data Items* (*DI*)**. It represents the protected data items in QFilter, such as tuples, attributes, or cells in a relation.
- ***Actions* (*A*)**. It encompasses the available aggregation functions in QFilter, namely "count", "sum", and "avg".
- ***Policies* (*P*)**. It includes all the access control policies associated with the data items in QFilter.

In this model, users possess identity attributes (e.g., name, affiliation, office number, job title, role, trust level), and ABAC policies are linked to data items to define access conditions based on users' identity attributes. A user whose attributes satisfy the ABAC policy associated with a data item is granted permission to perform a specific action (i.e., an aggregation function) on that data item.

Definition 1 (User Attribute (UA))**.** *A user attribute is defined as a pair of (aName, aType) such that* $\forall a \in UA : a = (aName, aType)$*, where aName is a unique attribute name and aType is a predefined data type in QFilter.*

In accordance with Definition 1, user attributes require predefined data types (e.g., integer, floating point, boolean, etc.) to ensure consistency and avoid any potential ambiguity or type mismatching in access control policies.

Example 2. *Three user attributes (Role, String), (Age, Integer), (Sex, Boolean) can be defined in QFilter based on Definition 1.*

Definition 2 (User Attribute Condition)**.** *A user attribute condition is defined in the form of cond: attrName compOp attrVal, where cond is a unique condition name, attrName is the name of an identity attribute of users, compOp is a comparison operator such as "=", ">=", "<=", ">", "<", and "!=", and attrVal is a value from the set of values in the predefined datatype aType that can be used by the identity attribute.*

Example 3. *Three user attribute conditions C1, C2, and C3 can be defined in QFilter based on Definition 2 as follows: C1: Role = "Banker", C2: Age > 18, and C3: Sex = 0.*

Definition 3 (User Attribute Condition Set)**.** *A user attribute condition set is defined in the form of condSet: cond | (cond boolOp condSet) | (condSet boolOp cond) | (condSet boolOp condSet), where condSet is a unique name for the condition set, cond is a unique condition name, boolOp is a boolean operator which can be "∧" or "∨", and "|" is a discriminator.*

Example 4. *User attribute condition set CS1: (C1 ∧ (C2 ∨ C3)) can be defined in QFilter based on Definition 3.*

Based on Definition 3, our proposed ABAC model allows user attribute conditions specified by the data owner to be combined as a set of conditions, enabling the specification of complex conditions. This feature supports the modeling of complex real-world situations.

Definition 4 (User Group)**.** *A user group is defined as a set of data users whose identity attributes satisfy the user attribute condition set in an ABAC policy.*

In our proposed ABAC model, each unique user attribute condition set is mapped to a unique user group (according to Definition 5).

Definition 5 (Mapping of a User Attribute Condition Set to a User Group)**.** *Given a set of user attribute condition set UACS =* $\{condSet_1, condSet_2, ..., condSet_t\}$*, a unique user group* G_i *can be automatically created by QFilter for a user attribute condition set* $condSet_i$*, where* $\forall 1 \leq i \leq t$*.*

Example 5. *User group* G_1 *can be automatically mapped to the user attribute condition set CS1 in Example 4 based on Definition 5.*

Definition 6 (ABAC Policy)**.** *An ABAC policy is defined as a triple (userGroup, dataItem, aggrFunc), where userGroup is a user group, each member of which has a set of*

identity attributes which satisfy a specific user attribute condition set, dataItem is a data item in relation R, and aggrFunc is an aggregation function supported by QFilter.

Example 6. *An ABAC policy can be defined as follows: (G_1, Balance, sum) in which a user who is a member of user group G_1 can perform the aggregation function sum on the data item Balance in the relation Account (Table 1).*

Based on Definition 6, our ABAC model only supports positive access authorizations, implying that access to a specific data item is denied by default.

4.2. Policy Attachment in QFilter

To attach ABAC policies to the outsourced relation *R*, new attributes representing these policies are added to the relation. For tuple-level access control, two additional attributes are introduced, one for the "count" aggregation function and another for the "sum" aggregation function. For attribute/cell-level access control, two attributes are introduced for each attribute in relation *R*, one for the "count" aggregation function and another for the "sum" aggregation function. Each value of the new attribute corresponds to a labeled ABAC policy, associated with a unique user group according to Definition 5.

It is important to note that our proposed ABAC model excludes access control policies for the aggregation function "avg". This decision is based on the observation that the result of aggregation function "avg" can be derived from the results of aggregation functions "sum" and "count". By relying on the ABAC policies for the aggregation functions "sum" and "count", QFilter prevents potential conflicts between different access control policies for the aggregation function "avg". Therefore, when a data user requests an aggregation SQL query including "avg" function, the corresponding ABAC policies for the aggregation functions "sum" and "count" are checked instead of explicitly checking access control policies for the aggression function "avg". Additionally, QFilter does not allow the setting of access control policies for specific data items (i.e., tuples, attributes, or cells) with different user groups for the aggregation functions "count" and "sum" in order to avoid conflicts among access control policies for the aggregation function "avg". However, it is permissible to set access control policies with different user groups for the aggregation function "sum" (or "count") when access to preform the aggregation function "count" (or "sum") for such data items are denied, as the aggregation function "avg" is not allowed in such cases.

To support the case that no ABAC policy is specified for a specific data item in relation *R*, QFilter uses a user group called *Min User Group*, which does not map to any user attribute condition set in the system.

Definition 7 (Min User Group)**.** *The user group G_0 is defined as the Min User Group if it is not mapped to any of the existing user attribute condition sets in QFilter.*

Based on Definition 7, if there is no ABAC policy for a data item in relation *R* (i.e., access denied for all data users), the value of the new attribute for this data item in relation *R* is set to G_0.

Example 7. *Tables 5 and 6 illustrate the policy attachment process for relation Account (Table 1), with tuple and attribute/cell access control granularity levels, respectively. As shown in Table 5, users in user group G_1 have access to the first tuple of relation R for performing the aggregation functions "count" and "sum". However, none of the users in G_1 can access the second and third tuples to preform an aggregation function, as the attributes Count and Sum are set to G_0 and G_2 for them, respectively. In Table 6, users in user group G_1 can access the first and third tuples of relation R for performing the aggregation functions "count" and "sum" on the Balance data item, as the attribute values Balance-Count and Balance-Sum are set to G_1. As shown in Table 6, the attributes Account No.-Sum and Account Type-Sum in all tuples of relation Account are set to G_0, as the "sum" aggregation function is not defined for the attributes Account No. and Account Type.*

Table 5
Tuple Level Policy Attachment

Account No.	Account Type	Balance (×1000$)	Count	Sum
1	checking	2	G_1	G_1
2	saving	3	G_0	G_0
3	checking	1	G_2	G_2

5. Data Outsourcing and Privacy Preserving Query Processing

In this section, we provide a detailed description of our proposed solution for data outsourcing and privacy-preserving query processing in QFilter. It utilizes an oblivious query rewriting and processing technique to tightly integrate the access control enforcement mechanism into the query processing workflow. The solution involves the following steps: 1) Creation and distribution of secret shares by the data owner, 2) Query submission and distribution by the data user, 3) Query rewriting and processing by non-communicating servers, and 4) Query result collection by the data user. We also consider the complexity of our proposed solution in terms of number of communication and computation rounds at both server and data user sides.

Table 6
Attribute or Cell Level Policy Attachment

Account No.	Account No.-Count	Account No.-Sum	Account Type	Account Type-Count	Account Type-Sum	Balance (×1000$)	Balance-Count	Balance-Sum
1	G_1	G_0	checking	G_1	G_0	2	G_1	G_1
2	G_2	G_0	saving	G_2	G_0	3	G_2	G_2
3	G_1	G_0	checking	G_1	G_0	1	$G1$	$G1$

5.1. Creation and Distribution of Secret-Shares

To create a set of secret-shares for relation R (i.e., including data and associated access control policies), we need to represent each value of relation R in a unary form as explained in Section 2.2. Example 8 shows how to represent different numerical values in a unary form.

Example 8. *Assume that a relation contains only numerical values. Generally, a numerical value can be represented by a unary array with 10 bits since we have only 10 numbers from '0' to '9' in decimal form. Hence, the number '1' can be represented as* $(1_1, 0_2, 0_3, ..., 0_{10})$*, where the subscript indicates the position of the numerical value; since '1' is the first number, the first bit in the unary array is one and others are zero. Similarly, '2' is* $(0_1, 1_2, 0_3, ..., 0_{10})$*, ..., '9' is* $(0_1, 0_2, ..., 0_8, 1_9, 0_{10})$*, and '0' is* $(0_1, 0_2, ..., 0_9, 1_{10})$*.*

This process can be followed in a similar way to represent other symbols. Example 9 shows how to represent different letters in the English alphabet in a unary form.

Example 9. *The English alphabet contains 26 letters. Each letter can be represented by a unary array with 26 bits. Hence, 'A' can be represented as* $(1_1, 0_2, ..., 0_{26})$ *since 'A' is the first letter and therefore, the first bit in the unary array is one and others are zero. 'B' can be represented as* $(0_1, 1_2, ..., 0_{26})$ *since 'B' is the second letter and therefore, the second bit in the unary array is one and others are zero, and so on.*

In the AA method, a data user can search for a b-bit string pattern without the need for inter-server communication. Both the data owner and data user employ a polynomial of degree one, resulting in a final polynomial degree of $2b$ due to secret-share multiplication during string matching. Solving this polynomial requires $2b+1$ secret-shares from different servers. However, in some cases it is possible that values in relation R can be mapped to a unary representation with fewer bits, offering greater efficiency and reducing the required secret-shares from non-communicating servers. Such mapping requires prior agreement between the data owner and data users, akin to marshaling in distributed systems. Example 10 illustrates this process for the *Account* relation.

Example 10. *Table 7 shows the output of unary representation of values in the relation Account including the tuple level policy attachment (Table 5). We only need to use 3 bits to represent the values of attributes Account-No., Balance, Count, and Sum in the unary representation form since each of them has only three different values. Moreover, we only need to use 2 bits to represent the values of attribute Account Type in the unary representation form since it has two different values.*

Table 7
Unary Representation of Account Relation

Account No.	Account Type	Balance (×1000$)	Count	Sum
100	01	010	100	100
010	10	001	001	001
001	01	100	010	010

By outsourcing the unary representation form of relation values, the data owner risks exposing the underlying data. To mitigate this, the data owner employs b polynomials of the same degree, where b represents the number of bits in the unary representation of a value. These polynomials generate b secret-shares for each specific value, which are then distributed to a designated server. Rather than transmitting the actual unary representation, the secret-shares are sent. Example 1 in Section 2.2 provides a demonstration of this process for the value of "checking" in the attribute *Account Type* of the relation *Account*.

Assume that R with n tuples and m attributes denoted by A_1, A_2, ..., and A_m is a relation which should be outsourced. In the case of tuple level policy attachment, two new attributes *Count* and *Sum* are added to relation R whose values specify the ABAC policy attached to each tuple of relation R. Therefore, relation R^t (t stands for tuple level policy attachment) with n tuples and $m+2$ attributes denoted by A_1, A_2, ..., A_m, *Count*, and *Sum* should be outsourced to a set of non-communicating servers. In the case of attribute/cell level policy attachment, two new attributes are added for each attribute in relation R whose values specify the ABAC policies associated with each attribute in relation R. Therefore, relation R^a (a stands for attribute level policy attachment) with n tuples and ($3 \times m$) attributes denoted by A_1, A_1^{Count}, A_1^{Sum}, A_2, A_2^{Count}, A_2^{Sum}, ..., A_m, A_m^{Count}, and A_m^{Sum} should be outsourced to a set of non-communicating servers. Now, assume that v_{ij} is the value of the ith tuple and jth attribute in relations R^t and R^a, and c is the number of non-communicating servers. Therefore, the data owner creates c secret-shares for the value v_{ij} (i.e., $S(v_{ij})$). The result of this step is c

secret-shared relations (i.e., R^t_1, R^t_2, ..., and R^t_c in the case of tuple level policy attachment and R^a_1, R^a_2, ..., and R^a_c in the case of attribute/cell policy attachment). Then, the pth secret-shared relation (i.e., R^t_p in the tuple level policy attachment and R^a_p in the attribute/cell level policy attachment) is outsourced to the pth server.

To rewrite the aggregation SQL query submitted by the data user u at the server side, all servers need the list of user groups of which the data user u is a member of to add new query conditions as access conditions in the WHERE clause of the submitted query. To provide this information, the data owner creates relation *userInfo* = (*Credential*, *User Group*) to store the values of credentials and user groups for each registered data user in QFilter. The values of attributes *Credential* and *User Group* are inserted into relation *userInfo* during the user registration process when a specific credential is created for the data user u and all the ABAC policies specified by the data owner are considered to find the list of user groups of which the data user u with a set of identity attributes is member of. However, outsourcing the values of attribute *User Group* in relation *userInfo* may infer some information about associated access control policies by revealing relationships between different user groups in the system. It is noted that servers need the exact values of attribute *Credential* in the process of user's authentication. To prevent the leakage of access control policies, the data owner creates c secret-shares for each value of attribute *User Group* and then creates c relations $userInfo_1$, $userInfo_2$, ..., and $userInfo_c$ for relation *userInfo* in such a way that the values of attribute *Credential* are unchanged and the values of attribute *User Group* are replaced by their corresponding secret-shares. Finally, the pth relation of *userInfo* (i.e., $userInfo_p$) is send to the pth server.

Example 11. *Table 8 (a) provides an example of the userInfo relation that needs to be outsourced to multiple non-communicating servers. It demonstrates that data users with specific credentials can belong to various user groups in our proposed ABAC model. Table 8 (b) displays the attributes Credential and User Group to be outsourced to the pth server. The notation $S(G_x)_p$ represents the secret-share of the xth user group (G_x) in the $userInfo_p$ relation.*

Table 8
Relations $userInfo$ and $userInfo_p$

Credential	User Group
c_1	G_1
c_1	G_3
c_2	G_2
c_3	G_1
c_3	G_2

(a) $userInfo$ Relation

Credential	User Group
c_1	$S(G_1)_p$
c_1	$S(G_3)_p$
c_2	$S(G_2)_p$
c_3	$S(G_1)_p$
c_3	$S(G_2)_p$

(b) $userInfo_p$ Relation

5.1.1. Discussion about Information Leakage

By outsourcing relation R as secret shares, the actual values of relation R and associated access control policies remain unknown to adversaries. Also, outsourcing the secret shared values of the attribute *User Group* in the *userInfo* relation does not reveal any information about the user groups to adversaries. It relies on the adversary's inability to collude with the majority of servers or access the communication channel between the data owner and individual servers during data outsourcing (as assumed in Section 3.2). Our solution employs different polynomials to generate secret shares for each occurrence of a specific value, ensuring that multiple occurrences of a value have distinct secret shares. Consequently, observing the secret-shared values does not reveal any information about relation R or the attribute *User Group* in relation *userInfo*. This mitigates the risk of frequency analysis and protects against *Frequent Count Attacks*. To conceal the actual number of user groups in relation *userInfo*, the data owner can introduce unused user groups and randomly assign registered data users to them. For instance, Table 8 (b) may include fake user groups to safeguard against the inference of user groups and access policies. It is important to note that the actual value of the attribute *Credential* cannot infer any information about the access control policies associated with the data items in relation R, as *Credential* is solely used to authenticate the data users at the server side.

5.2. Query Submission and Distribution

Our proposed solution supports both the simple and multi dimensional aggregation SQL queries as shown in Table 9. In the following, we explain how a data user can submit and distribute an aggregation SQL query to the set of non-communicating servers.

1. Simple aggregation SQL queries: Assume that the data user u wishes to submit the aggregation SQL query Q_1 in the form of "select $\alpha(A_i)$ from R" to the servers. This query will be distributed to every individual server without any changes.
2. Multi-dimensional aggregation SQL queries: In the case that the data user u wishes to submit the aggregation SQL query Q_2 in the form of "select $\alpha(A_i)$ from R where $(A_k = v_k)$ OP ... OP $(A_l = v_l)$", the actual values in the query conditions (e.g., v_k and v_l) should be represented in the unary form. Then, a set of c secret-shares should be created for each bit of them by the data user u, where c is the number of non-communicating servers. Such a process is explained in Example 1 in Section 2.2. Assume that the pth set of secret-shares for all bits of the values of v_k and v_l is denoted as $S(v_k)_p$ and $S(v_l)p$, respectively. Then, these secret-shares

Table 9
Types of Supported Aggregation Queries in Our Proposed Approach

Aggregation SQL Query Type	Query Format
Simple Aggregation SQL Queries	select $\alpha(A_i)$ from R
Multi-Dimensional Aggregation SQL Queries	select $\alpha(A_i)$ from R where $(A_k{=}v_k)$ OP ... OP $(A_l{=}v_l)$

Note: α can be "count", "sum", or "avg" and OP can be "$\wedge$" or "$\vee$" operator.

are replaced by the actual values in the query conditions to hide the query pattern from every individual server. Thus, the query Q_{2p} in the form of "select $\alpha(A_i)$ from R where $(A_k = S(v_k)_p)$ OP ... OP $(A_l = S(v_l)_p)$" will be distributed to the pth server.

5.2.1. Discussion about Information Leakage

By employing different polynomials to create a set of secret shares for each value in each query condition, our proposed solution hides the query pattern of the submitted aggregation SQL query from adversaries. This applies specifically to multi-dimensional aggregation SQL queries, where the query pattern remains undisclosed. Conversely, in the case of simple aggregation SQL queries without any query conditions, there is no need to worry about query patterns being obvious. However, it is worth mentioning that adversaries can acquire information about the submitted query, such as the type of query (i.e., simple or multi-dimensional), the specific type of aggregation function used (i.e., "count," "sum," or "avg"), the attribute to which the aggregation function is applied, and the total number of conjunctive or disjunctive equality query conditions. Nevertheless, this information alone does not enable adversaries to ascertain the exact values of outsourced data, associated access control policies, query conditions, or query results, as they are all represented in the form of secret shares.

5.3. Query Rewriting and Processing

In our proposed solution, every individual server performs the tasks of query rewriting and query processing without the need for communicating with other entities.

5.3.1. Query Rewriting

When an aggregation SQL query is submitted by the data user u on relation R, it is necessary for QFilter to check the set of ABAC policies attached to relation R to restrict the query result to the only data items which the data user u has access to. In the following, we explain in detail how QFilter obliviously rewrites an aggregation SQL query at the server side.

1. Simple aggregation SQL queries: Assume that the aggregation SQL query Q_p = "select $\alpha(A_i)$ from R" is sent from the data user u to the pth server and F_p is relation R^t_p in the case of tuple level access control and relation R^a_p in the case of attribute/cell level access control outsourced on the pth server. This query is rewritten as follows: Q_p = "select $\alpha(A_i)$ from F_p where $(\beta = S(UG_1)_p) \vee (\beta = S(UG_2)_p)$ $\vee$ ... $\vee$ $(\beta = S(UG_q)_p)$", where β is the attribute α in relation R^t_p or attribute A^α_i in relation R^a_p, and $S(UG_x)_p$ is the secret-share of the xth user group UG $(\forall 1 \leq x \leq q)$ in relation $userInfo_p$ which the data user u is a member of. In the process of query rewriting, we need to add a query condition in the WHERE clause of the query $Q^{'}_p$ for each user group which the data user u is member of since the data user u can be a member of different user groups and the set of ABAC policies are mapped into user groups in the system.
2. Multi-dimensional aggregation SQL queries: Assume that the aggregation SQL query Q_p = "select $\alpha(A_i)$ from R where $(A_k = S(v_k)_p)$ OP ... OP $(A_l = S(v_l)_p)$" is sent from the data user u to the pth server and F_p is relation R^t_p in the case of tuple level access control and relation R^a_p in the case of attribute/cell level access control outsourced on the pth server. This query is rewritten as follows: $Q^{'}_p$ = "select $\alpha(A_i)$ from F_p where $(A_k = S(v_k)_p)$ OP ... OP $(A_l = S(v_l)_p) \wedge ((\beta = S(UG_1)_p) \vee (\beta = S(UG_2)_p)$ $\vee$ ... $\vee$ $(\beta = S(UG_q)_p))$", where β is the attribute α in relation R^t_p or attribute A^α_i in relation R^a_p, and $S(UG_x)_p$ is the secret-share of the xth user group UG $(\forall 1 \leq x \leq q)$ in relation $userInfo_p$ which the data user u is a member of. In this case, a set of query conditions is added in the WHERE clause of the query $Q^{'}_p$ using "$\wedge$" operator. These query conditions are specified for the user groups which the data user u is member of and each two query conditions are combined together using "$\vee$" operator.

Example 12. *Assume that the data user u_1 with the credential c_1 wishes to submit the query Q in the form of "Select count(Balance) from Account where (Account Type = "checking")" on relation Account (in Table 5). Such a query is distributed to the pth server in the form of "Select count(Balance) from Account where (Account Type = $S(checking)_p$)". Next, this query is rewritten by the pth server as follows: "Select count(Balance) from Ac-*

count where (Account Type = $S(checking)_p$) $\wedge$ (($Count = S(G_1)_p$)$\vee$($Count = S(G_3)_p$)) since the data user u_1 with the credential c_1 is a member of user groups G_1 and G_3 (refer to Table 8).

5.3.2. Query Processing

In query processing, the pth server ($\forall 1 \leq p \leq c$) executes the pth aggregation SQL query $Q_p^{'}$ over the pth outsourced relation F_p (i.e., relation R_p^t in the case of tuple level access control and relation R_p^a in the case of attribute/cell level access control) and filters the data items that do not satisfy the query conditions in the query $Q_p^{'}$. To find the result of string matching (i.e., which can be "0" or "1" in the form of secret-share) for each query condition in the query $Q_p^{'}$, the operator $\odot$ as the string matching operator is used by the pth server including a bit-wise multiplication followed with an addition over all values of bits of secret-shares in the unary representation form. It is defined as follows:

$$Result_y^z = \begin{cases} S(v(\alpha)_z)_p \odot S(v_y)_p & \text{if } \beta = \alpha \\ S(v(A_y^\alpha)_z)_p \odot S(v_y)_p & \text{if } \beta = A_i^\alpha \\ S(v(A_y)_z)_p \odot S(v_y)_p & \text{otherwise} \end{cases}$$

where $S(v(\alpha)_z)_p$ is the secret-share of attribute α in the zth tuple of relation F_p (i.e., R_p^t), $S(v(A_y^\alpha)_z)_p$ is the secret-share of attribute A_y^α in the zth tuple of relation F_p (i.e., R_p^a), $S(v(A_y)_z)_p$ is the secret-share of attribute A_y in the zth tuple of relation F_p (i.e., R_p^t or R_p^a), and $S(v_y)_p$ is the secret-share of the corresponding query condition in the query $Q_p^{'}$. It should be noted that α is the aggregation function in the query $Q_p^{'}$. Table 4 in Example 1 shows how this operator can be used by every individual server for obliviously searching the string pattern "checking" over outsourced data.

The results of string matching can be used to compute the result of a specific aggregation function in the form of secret-share at the pth server. Such a process varies depending on the type of aggregation functions. In the following, we explain in detail how to exploit the result of string matching to process aggregation functions.

1. Count Function: The result of an aggregation SQL query including "count" function can be computed by the pth server using the following operation:

$$output = \sum_{z=1}^{n}(((Result_{A_k}^z \circledast ...) \circledast Result_{A_l}^z)) \wedge$$
$$(((Result_{\beta_1}^z \vee Result_{\beta_2}^z) \vee ... \vee Result_{\beta_q}^z))$$

   where $\circledast$ is OP (i.e, $\wedge$ or $\vee$), A_k, ..., A_l are the set of attributes in the WHERE clause of query Q_p (and $Q_p^{'}$), and β_i is the ith β in the WHERE clause of query $Q_p^{'}$ ($\forall 1 \leq i \leq q$). To capture the operation "$\wedge$" and compute the final result of ($Result_i^z \wedge Result_j^z$) in the form of secret-share for each tuple z, the pth server executes the following computation:

$$Result_{ij}^z = Result_i^z \times Result_j^z$$

   To capture the operation "$\vee$" and compute the final result of ($Result_i^z \vee Result_j^z$) in the form of secret-share for each tuple z, the pth server executes the following computation:

$$Result_{ij}^z = Result_i^z + Result_j^z - Result_i^z \times Result_j^z$$

   The correctness of the output of "count" function can be described as if the zth tuple has "0" in the form of secret-share as a comparison resultant of $(((Result_{A_k}^z \circledast ...) \circledast Result_{A_l}^z))$ or $(((Result_{\beta_1}^z \vee Result_{\beta_2}^z) \vee ... \vee Result_{\beta_q}^z))$, it will produce "0" in the secret-share form as the result of this operation for the zth tuple; therefore, the zth tuple will not counted as the result of this operation. Thus, the correct occurrences over all tuples that satisfy the query's WHERE clause are counted as the result of this operation.
2. Sum Function: The result of an aggregation SQL query including "sum" function can be computed by the pth server using the following operation:

$$output = \sum_{z=1}^{n} S(v(A_i)_z)_p \times (((Result_{A_k}^z \circledast ...) \circledast Result_{A_l}^z)$$
$$\wedge (((Result_{\beta_1}^z \vee Result_{\beta_2}^z) \vee ... \vee Result_{\beta_q}^z)))$$

   where A_i is the attribute which the aggregation function "sum" is applied on, $S(v(A_i)_z)_p$ is the secret-share of the attribute A_i of zth tuple of relation F_p, $\circledast$ is OP (i.e, $\wedge$ or $\vee$), A_k, ..., A_l are the set of attributes in the WHERE clause of query Q_p (and $Q_p^{'}$), and β_i is the ith β in the WHERE clause of query $Q_p^{'}$ ($\forall 1 \leq i \leq q$). The process of computation to find the final results of ($Result_i^z \vee Result_j^z$) and ($Result_i^z \wedge Result_j^z$) for each tuple z is similar to this process in "count" function. In addition, the argument for the correctness of "sum" operation is similar to the correctness of the operation "count".
3. Avg Function: The result of an aggregation SQL query including "avg" function can be computed by dividing the result of corresponding aggregation SQL query including "sum" function by the result of corresponding aggregation SQL query including "count" function. In this case, every individual server sends the query results for these two queries to the data user u. Upon arrival of

Table 10
Execution of Multi-Dimensional SQL Query including "Count" Function

Account Type	Value of the First Condition	$Result^{k}_{AccountType}$	Count	Value of the Second Condition	$Result^{k}_{Count1}$	Count	Value of the Third Condition	$Result^{k}_{Count2}$
$S(checking)_p$	$S(checking)_p$	$S(1)$	$S(G_1)_p$	$S(G_1)_p$	$S(1)$	$S(G_1)_p$	$S(G_3)_p$	$S(0)$
$S(saving)_p$	$S(checking)_p$	$S(0)$	$S(G_0)_p$	$S(G_1)_p$	$S(0)$	$S(G_0)_p$	$S(G_3)_p$	$S(0)$
$S(checking)_p$	$S(checking)_p$	$S(1)$	$S(G_2)_p$	$S(G_1)_p$	$S(0)$	$S(G_2)_p$	$S(G_3)_p$	$S(0)$

the query results, the data user u utilizes them to compute the result of aggregation SQL query including "avg" function.

Example 13. *Assume that the query $Q^{'}_{p}$ in the form of "Select count(Balance) from Account where (Account Type = $S(checking)_p$) $\wedge$ (($Count = S(G_1)_p$) $\vee$ ($Count = S(G_3)_p$))" is the output of the query rewriting process by the pth server (Refer to Example 12). Table 10 shows how do the pth server executes this query using string matching based operations. The results of string matching are used to compute the result of the query $Q^{'}_{p}$ at the pth server using the following operation:*

$$output = \sum_{z=1}^{3}(Result^{z}_{AccountType} \wedge (Result^{z}_{Count1} \vee Result^{z}_{Count2})) =$$

$$(S(1)\wedge(S(1)\vee S(0)))+(S(0)\wedge(S(0)\vee S(0)))+(S(1)\wedge(S(0)\vee S(0))) =$$

$$S(1) + S(0) + S(0) = S(1).$$

Note that this process will be performed on the unary representation form of secret-shares. However, we show cleartext values for simple explanation here.

5.3.3. Discussion about Information Leakage

By rewriting the aggregation SQL query Q and introducing a set of q query conditions in the WHERE clause, where q represents the number of user groups the data user belongs to in the relation $userInfo_i$, the query pattern of Q remains hidden from adversaries. The inclusion of secret shares of the relation $userInfo_i$ in query conditions prevents adversaries from inferring any information about user groups or their associated access control policies. Additionally, the total number of user groups to which the data user belongs can be obscured by the presence of fake user groups in the relation $userInfo_i$. By employing string matching-based operators for computing the aggregation functions "count" and "sum" in the query Q, QFilter conceals the access pattern during query processing and access authorization checking. This ensures that the identity of tuples satisfying or being filtered out from the query Q remains hidden from adversaries since these operators include bit-wise multiplication followed by addition over all values of bits of secret-shares of attributes that appear in the query condition for each tuple. Therefore, string matching-based operators obliviously search for string pattern matching. Moreover, the actual values and sizes of query results are masked from adversaries, as the output of Q is provided in the form of secret shares with an identical number of bits. Additionally, it is assumed that there are no conflicts among the access control policies specified for the aggregation functions "sum" and "count" in each data item. Therefore, utilizing string matching-based operators to compute these aggregation functions, instead of computing the aggregation function "avg", does not leak any sensitive information. This assumption is based on the premise that either the aggregation functions "count" and "sum" are allowed to be performed for a data item by setting the same user group for that data item, or at least one of them is denied by setting the user group of that data item as G_0. It should be noted that although adversaries may obtain some information about the rewritten query such as the exact type of submitted query (i.e., simple and multi-dimensional queries), the specific type of aggregation function used in the query (i.e., "count", "sum", and "avg"), the attribute to which the aggregation function is applied, and the total number of conjunctive or disjunctive equality query conditions, this information alone cannot help adversaries learn about the exact value of outsourced data, associated access control policies, query conditions, and query results. This is because they are represented in a secret-shared form.

5.4. Query Result Collection

After receiving the query results in the form of secret shares with an identical number of bits from different servers, the data user performs the Lagrange interpolation operation on the received results to obtain the final answer for the submitted aggregation SQL query. This process is explained in Example 1 in Section 2.2.

5.5. Complexity of Our Approach

Table 11 presents the complexity of our proposed solution for processing various types of aggregation SQL queries. As depicted in Table 11, our solution entails a single communication round between the data user and every individual server, as well as one computation round to scan the tuples during the server-side query processing for all types of aggregation SQL queries. To interpolate the query results at the data user side, only one computation round is needed for aggregation SQL queries including the "count" or "sum" functions. For aggregation SQL queries including the "avg" function,

Table 11
Complexity of Our Proposed Approach

Aggregation SQL Query	Computation Rounds at the Server Side	Communication Rounds	Computation Rounds at the Data User Side
Aggregation Queries including "count" Function	1	1	1
Aggregation Queries including "sum" Function	1	1	1
Aggregation Queries including "avg" Function	1	1	2

it requires two computation rounds (one for "sum" and one for "count") to perform the Lagrange interpolation operation.

6. Experimental Evaluation

In this section, we evaluate the overhead of QFilter for data outsourcing and privacy preserving query processing through preliminary experiments.

6.1. Setup

We implemented QFilter using JAVA programming language and conducted our experiments on a machine equipped with a 3.60 GHz Intel® Core™ i7-7700 CPU and 16 GB of RAM. This machine was utilized by the data owner, data user, and every server involved in our experiments.

For generating datasets, we employed the LINEITEM relation from the TPC-H benchmark. To prevent an adversary to learn about the distribution of values in the LINEITEM relation, we added a set of zeros to the left side of the unary representation of values in such a way that all values contain identical bits. To create secret shares for the values in the LINEITEM relation, we selected different polynomials of degree 1 with randomly generated coefficients. Table 12 provides a comprehensive list of the parameters used in our experiments, along with their corresponding values.

Table 12
List of Parameters and Their Values

Parameter	Value
Number of Servers	5, 15 (default), 25
Number of Tuples	100K (default), 250K, 500K
Number of Attributes	3, 4 (default), 5, 7

In our setting, we randomly selected 50% of the data items in the LINEITEM relation as accessible data items and assigned them to the user group G_1. Conversely, all inaccessible data items were assigned to the default user group G_0 (i.e., the *Min User Group*). We made the assumption that the data user submitting an aggregation SQL query belongs to the user group G_1.

We utilized a set of aggregation SQL queries in our experiments, named in the form of F-XYZ. Here, F denotes the type of aggregation function in the query (*C* for "Count", *S* for "Sum", and *A* for "Avg"), XY represents the type of queries (*SI* for "Simple", *CE* for "Conjunctive Equality", and *DE* for "Disjunctive Equality"), and Z denotes the number of query conditions in the WHERE clause of the query (i.e., 0, 2, and 4).

In [42], it is demonstrated that the utilization of the AA method [48] for privacy-preserving query processing has outperformed existing approaches. Therefore, our main focus here is to evaluate the overhead of QFilter for data outsourcing and privacy preserving query processing in two cases: tuple-level and attribute-level policy attachments.

6.2. Experimental Results

In this section, we present the preliminary experimental results to demonstrate the performance overhead of QFilter for data outsourcing and privacy-preserving query processing.

6.2.1. Computation at the Data Owner Side

Table 13 displays the average time required to create secret shares for the LINEITEM relation, as well as the total size of the generated dataset. In these measurements, we considered a scenario where the number of attributes was 4, the number of servers was 15, and no access policy was attached to the relation.

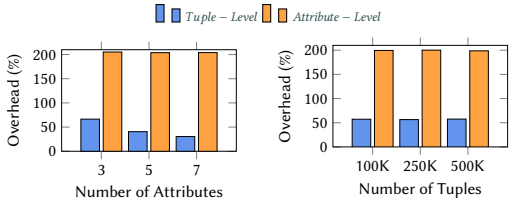
Table 13
Average Time and Total Size for Creating Secret Shares

No. of Tuples	Avg Time	Time for Each Attribute	Total Size	Size of Each Attribute
100K	12.7832 s	2.13×10^{-6} s	94.16 MB	16 Byte
250K	32.1551 s	2.14×10^{-6} s	235.41 MB	16 Byte
500K	64.5962 s	2.15×10^{-6} s	470.82 MB	16 Byte

As observed in Table 13, it is evident that both the average time to create secret shares and the total size of the generated dataset increase with an increasing number of tuples in the LINEITEM relation. QFilter requires 16 bytes to store each value of the LINEITEM relation in unary representation form due to the uniform bit length of all values within the LINEITEM relation. However, the creation of secret shares can still be accomplished within a short period of time (approximately 2.15 microseconds for each attribute of the LINEITEM relation). Similar results are obtained when the number of attributes increases. It should be noted that attributes can be added to

the LINEITEM relation to specify access policies. However, the number of attributes to be added depends on the type of policy attachment, whether it is at the tuple level or the attribute/cell level.

Figure 2 shows the impact of number of attributes and tuples on the computational overhead of QFilter to create secret shares for both types of policy attachments. As shown in Figure 2a, the computational overhead to create secret shares for the tuple-level policy attachment decreases when the number of attributes in the LINEITEM relation increases. The reason is that only two new attributes should be added to the LINEITEM relation for the aggregation functions "count" and "sum" in our proposed approach. However, increasing the number of attributes in the case of attribute-level policy attachment does not have any effect on the computational overhead. This is due to the fact that two new attributes (for "count" and "sum" functions) should be added in our proposed solution to specify ABAC policies for each attribute of the LINEITEM relation. It is also clear from Figure 2b that the number of tuples does not have any effect on the computational overhead for both types of policy attachments. This is because new attributes should be added for each tuple of the LINEITEM relation in the process of policy attachment.



(a) Impact of No. of Attributes (b) Impact of No. of Tuples

Figure 2: Computational Overhead at the Data Owner Side

Based on the experimental results shown here, it is obvious that there is a trade-off between enforcing finer granularity of access control and the computational overhead for creating secret shares. QFilter provides finer granularity to specify ABAC policies in the case of attribute-level policy attachment, but it imposes more computational overhead compared to the tuple-level policy attachment for creating secret shares at the data owner side.

6.2.2. Computation at the Server Side

Figure 3 illustrates the computation time required to process different aggregation SQL queries for both types of policy attachments at the server side. As depicted in Figure 3, the computation time for processing SQL queries including the aggregation function "sum" is longer compared to SQL queries including the aggregation function "count". This is due to the need for an additional multiplication operation for each tuple in the outsourced relation. Furthermore, the computation time for processing SQL queries including the aggregation function "avg" is approximately the same as the computation time for processing the corresponding SQL query with the aggregation function "sum". This is because the process of computing the result of the aggregation function "avg" at the server side is accomplished by simultaneously computing the results of the corresponding aggregation SQL queries, including the aggregation functions "sum" and "count". Since the "sum" function has more overhead compared to the "count" function, the overall computation times for processing the aggregation functions "sum" and "avg" remain similar. We also observed that as the number of query conditions increases, the computation time also increases, primarily due to the increased number of multiplications. Additionally, the computation time for processing aggregation SQL queries with attribute-level policy attachment is always longer than that with tuple-level policy attachment. This difference can be attributed to the longer data fetching and query processing time in the case of attribute-level policy attachment.

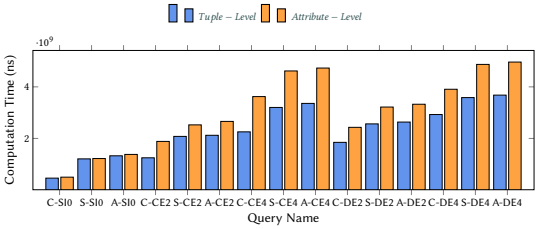


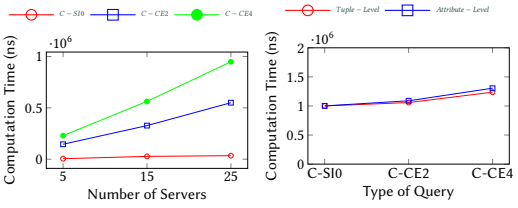
Figure 3: Computation Time for Processing Aggregation SQL Queries

Based on the experimental results presented here, it is apparent that the computational overhead for processing an aggregation SQL query at the server side can vary depending on several factors. These factors include the type of query, the specific aggregation function used, the number of query conditions, and the type of policy attachment.

6.2.3. Computation at the Data User Side

Figure 4a illustrates the computation time for creating secret shares of the values of query conditions for aggregation SQL queries *C-SI0*, *C-CE2*, and *C-CE4* at the data user side, considering varying numbers of servers. The computation time increases with a higher number of servers and query conditions due to additional computations required. However, the process can still be completed quickly. Figure 4b displays the computation time for the Lagrange interpolation operation on received

results from different servers to obtain the final answer for aggregation SQL queries *C-SI0*, *C-CE2*, and *C-CE4* for both types of policy attachments. As shown in Figure 4b, the computation time at the data user side is approximately the same for both types of policy attachments and different types of aggregation SQL queries since the total number of received results from different servers remains constant. It is worth noting that similar results were observed for other aggregation SQL queries.



(a) Query Distribution (b) Result Collection

Figure 4: Computation Time at the Data User Side

Based on the experimental results presented here, it is clear that the computation time for both creating secret shares of the values of query conditions in an aggregation SQL query and performing the Lagrange interpolation operation on received results at the data user side is negligible. This indicates that data users can utilize devices with limited resources to submit aggregation SQL queries and obtain the query results using QFilter.

7. Related Works

In theory, it is possible to utilize Fully Homomorphic Encryption (FHE) [20, 21, 22, 23, 24] to perform arbitrary query processing operations on a relation, but it is extremely complex and cannot practically support various types of queries [11, 10, 53]. By contrast, QFilter employs Shamir's secret sharing scheme which can efficiently process aggregation SQL queries over secret-shared data.

Several proposals have been suggested to improve the performance of FHE by utilizing specific hardware such as GPUs [54, 55, 56, 57]. However, this technique cannot be applied to low-cost hardware. In contrast, QFilter can be run on any hardware. Another solution to overcome the limitations of FHE is to employ Intel SGX as a hardware-assisted Trusted Execution Environment (TEE), which offers high computational efficiency, generality, and flexibility [58, 59, 60, 61]. However, this solution exposes access patterns due to side-channel attacks (such as cache timing [62, 63, 64], branch shadowing [65], and page fault attacks [66, 67]) on Intel SGX. In contrast, QFilter can obliviously evaluate aggregation SQL queries over secret-shared data without revealing access patterns and query patterns.

Attribute Based Encryption (ABE) is a public key cryptographic technique that achieves data confidentiality, data privacy, and access control in data outsourcing [68, 69]. It is mainly classified into two types: Key-Policy ABE (KP-ABE) and Ciphertext-Policy ABE (CP-ABE). In KP-ABE solutions [31, 32, 33, 34], the ciphertext is based on user attributes, and the user's secret keys are based on access policies, while in CP-ABE solutions [35, 36, 37], the ciphertext is based on access policies, and the user's secret keys are based on user attributes. However, these solutions may inadvertently expose user attributes and access policies to the public and are vulnerable to inference attacks [38, 39]. In contrast, QFilter can protect both the privacy of data users and the privacy of access policies.

8. Conclusion and Future Works

In this paper, we proposed QFilter to integrate an Attribute-Based Access Control (ABAC) model with aggregation query processing. Our proposed ABAC model was able to specify complex access control policies at the tuple, attribute, or cell level of the outsourced relation. QFiler used an oblivious query rewriting technique to add new query conditions for access authorization checking during query processing at the server side. To obliviously process aggression SQL queries (i.e., "count", "sum", and "avg" having single-dimensional, conjunctive (using "AND"), and disjunctive (using "OR") equality query conditions) over secret-shared data, QFilter used a set of string matching-based operators to compute query results without the need for communication between servers. We showed that QFilter was able to protect the privacy of outsourced data, its associated access control policies, query conditions, and query results from the *honest-but-curious* adversaries. Experimental results demonstrated that QFilter had lower overhead in the case of tuple level policy attachment and could be used for real-world applications.

In our future plans for extending QFilter, we aim to explore various directions, which include the following:

1. Design algorithms to process aggregation functions "min" and "max" in an aggregation SQL query by combining an order-preserving secret sharing (OP-SS) scheme with Shamir's secret sharing scheme. This combination will allow for the secure distribution of data while preserving the order of the secret-shared values.
2. Design an algorithm to process aggregation SQL queries with "group-by" and "having" clauses in an efficient and privacy-preserving manner without knowing the unique values of the attribute on which the group-by query is executed on.

3. Design an algorithm to create a set of bins over the domain of the attribute values and organize these bins into an index tree, which can be used to process range queries over secret shared data.
4. Design algorithms to process complex aggregation SQL queries, including multiple aggregation functions with inequality conditions in the query.
5. Conduct a comprehensive experiment to evaluate the efficiency and scalability of QFilter compared to existing approaches for privacy-preserving query processing.

Acknowledgments

This work was partially funded by the BMWK project SafeFBDC (01MK21002K), the National Research Center ATHENE, and the BMBF project TrustDBle (16KIS1267). We also want to thank hessian.AI at TU Darmstadt as well as DFKI Darmstadt for the support.

References

[1] J. Li, D. Lin, A. C. Squicciarini, J. Li, C. Jia, Towards privacy-preserving storage and retrieval in multiple clouds, IEEE Transactions on Cloud Computing 5 (2017) 499–509.

[2] M. Li, S. Yu, K. Ren, W. Lou, Y. T. Hou, Toward privacy-assured and searchable cloud data storage services, IEEE Network 27 (2013) 56–62.

[3] Y. Mansouri, A. N. Toosi, R. Buyya, Data storage management in cloud environments: Taxonomy, survey, and future directions, ACM Computing Surveys 50 (2018) 1–51.

[4] J. Wei, W. Liu, X. Hu, Secure data sharing in cloud computing using revocable-storage identity-based encryption, IEEE Transactions on Cloud Computing 6 (2018) 1136–1148.

[5] C. Ge, W. Susilo, Z. Liu, J. Xia, P. Szalachowski, L. Fang, Secure keyword search and data sharing mechanism for cloud computing, IEEE Transactions on Dependable and Secure Computing 18 (2021) 2787–2800.

[6] C. Wang, N. Cao, K. Ren, W. Lou, Enabling secure and efficient ranked keyword search over outsourced cloud data, IEEE Transactions on Parallel and Distributed System 23 (2012) 1467–1479.

[7] R. S. Sandhu, P. Samarati, Access control: principle and practice, IEEE communications magazine 32 (1994) 40–48.

[8] I. Indu, P. M. R. Anand, V. Bhaskar, Identity and access management in cloud environment: Mechanisms and challenges, Engineering science and technology, an international journal 21 (2018) 574–588.

[9] S. Rizvi, A. Mendelzon, S. Sudarshan, P. Roy, Extending query rewriting techniques for fine-grained access control, in: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, ACM Press, New York, NY, USA, 2004, p. 551–562.

[10] M. I. Sarfraz, M. Nabeel, J. Cao, E. Bertino, Dbmask: Fine-grained access control on encrypted relational databases, Transactions on Data Privacy 9 (2016) 187–214.

[11] M. I. Sarfraz, M. Nabeel, J. Cao, E. Bertino, Dbmask: Fine-grained access control on encrypted relational databases, in: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, ACM Press, New York, NY, USA, 2015, pp. 1–11.

[12] J.-G. Lee, K.-Y. Whang, W.-S. Han, I.-Y. Song, The dynamic predicate: integrating access control with query processing in xml databases, The VLDB Journal 16 (2007) 371–387.

[13] M. Mirabi, H. Ibrahim, L. Fathi, N. I. Udzir, A. Mamat, A dynamic compressed accessibility map for secure xml querying and updating, Journal of Information Science & Engineering 31 (2015) 59–93.

[14] M. Mirabi, H. Ibrahim, N. I. Udzir, A. Mamat, A compact bit string accessibility map for secure xml query processing, Procedia Computer Science 10 (2012) 1172–1179.

[15] B. Luo, D. Lee, W.-C. Lee, P. Liu, Qfilter: fine-grained run-time xml access control via nfa-based query rewriting, in: Proceedings of the 13 ACM international conference on Information and knowledge management, ACM Press, New York, NY, USA, 2004, p. 543–552.

[16] R. A. Popa, C. M. S. Redfield, N. Zeldovich, H. Balakrishnan, Cryptdb: Protecting confidentiality with encrypted query processing, in: Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11), ACM Press, New York, NY, USA, 2011, pp. 85–100.

[17] D. T. T. Anh, A. Datta, Streamforce: outsourcing access control enforcement for stream data to the clouds, in: Proceedings of the 4th ACM conference on Data and application security and privacy, ACM Press, New York, NY, 2014, p. 13–24.

[18] C. Thoma, A. J. Lee, A. Labrinidis, Polystream: Cryptographically enforced access controls for outsourced data stream processing, in: Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies, ACM Press, New York, NY, USA, 2016, p. 227–238.

[19] J. Li, J. Li, Z. Liu, C. Jia, Enabling efficient and secure data sharing in cloud computing, Concurrency and Computation: Practice and Experience 26 (2014) 1052–1066.

[20] M. Mani, Enabling secure query processing in the

cloud using fully homomorphic encryption, in: Proceedings of the Second Workshop on Data Analytics in the Cloud, ACM Press, New York, NY, USA, 2013, pp. 36–40.

[21] M. Kim, H. T. Lee, S. Ling, B. H. M. Tan, H. Wang, Private compound wildcard queries using fully homomorphic encryption, IEEE Transactions on Dependable and Secure Computing 16 (2019) 743–756.

[22] D. Boneh, C. Gentry, S. Halevi, F. Wang, D. J. Wu, Private database queries using somewhat homomorphic encryption, in: Proceedings of International Conference on Applied Cryptography and Network Security, Springer-Verlag, Berlin, Germany, 2013, pp. 102–118.

[23] M. Kim, H. T. Lee, S. Ling, H. Wang, On the efficiency of fhe-based private queries, IEEE Transactions on Dependable and Secure Computing 15 (2018) 357–363.

[24] B. H. M. Tan, H. T. Lee, H. Wang, S. Ren, K. M. M. Aung, Efficient private comparison queries over encrypted databases using fully homomorphic encryption with finite fields, IEEE Transactions on Dependable and Secure Computing 18 (2021) 2861–2874.

[25] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, P. Samarati, Balancing confidentiality and efficiency in untrusted relational dbmss, in: Proceedings of the 10th ACM conference on computer and communications security, ACM Press, New York, NY, USA, 2003, p. 93–102.

[26] S. Wang, D. Agrawal, A. E. Abbadi, A comprehensive framework for secure query processing on relational data in the cloud, in: Workshop on Secure Data Management, Springer-Verlag, Berlin, Germany, 2011, p. 52–69.

[27] M. Nabeel, E. Bertino, Privacy preserving delegated access control in public clouds, IEEE Transactions on Knowledge and Data Engineering 26 (2014) 2268–2280.

[28] M. Nabeel, E. Bertino, Privacy preserving delegated access control in the storage as a service model, in: Proceedings of the IEEE 13th International Conference on Information Reuse & Integration, IEEE Computer Society, Washington, DC, USA, 2012, pp. 645–652.

[29] B. Hore, S. Mehrotra, G. Tsudik, A privacy-preserving index for range queries, in: Proceedings of the 13th international conference on very large data bases, Springer-Verlag, Berlin, Germany, 2004, p. 720–731.

[30] A. Sahai, B. Waters, Fuzzy identity-based encryption, in: Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer-Verlag, Berlin, Germany, 2005, p. 457–473.

[31] V. Goyal, O. Pandey, A. Sahai, B. Waters, Attribute-based encryption for fine-grained access control of encrypted data, in: Proceedings of the 13th ACM conference on computer and communications security, ACM Press, New York, NY, USA, 2006, p. 89–98.

[32] R. Ostrovsky, A. Sahai, B. Waters, Attribute-based encryption with non-monotonic access structures, in: Proceedings of the 14th ACM conference on computer and communications security, ACM Press, New York, NY, USA, 2007, p. 195–203.

[33] A. Lewko, A. Sahai, B. Waters, Revocation systems with very small private keys, in: Proceedings of the 2010 IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 2010, pp. 273–285.

[34] N. Attrapadung, B. Libert, E. de Panafieu, Expressive key-policy attribute-based encryption with constant-size ciphertexts, in: Proceedings of the International Workshop on Public Key Cryptography, Springer-Verlag, Berlin, Germany, 2011, p. 90–108.

[35] J. Bethencourt, A. Sahai, B. Waters, Ciphertext-policy attribute-based encryption, in: Proceedings of the IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 2007, pp. 321–334.

[36] L. Cheung, C. Newport, Provably secure ciphertext policy abe, in: Proceedings of the 14th ACM conference on Computer and communications security, ACM Press, New York, NY, USA, 2007, p. 456–465.

[37] X. Liang, Z. Cao, H. Lin, D. Xing, Provably secure and efficient bounded ciphertext policy attribute based encryption, in: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, ACM Press, New York, NY, USA, 2009, p. 343–352.

[38] J. Hao, J. Liu, H. Wang, L. Liu, M. Xian, X. Shen, Efficient attribute-based access control with authorized search in cloud storage, IEEE Access 7 (2019) 182772–182783.

[39] K. Yang, Q. Han, H. Li, K. Zheng, Z. Su, X. Shen, An efficient and fine-grained big data access control scheme with privacy-preserving policy, IEEE Internet of Things Journal 4 (2017) 563–571.

[40] S. Dolev, P. Gupta, Y. Li, S. Mehrotra, S. Sharma, Privacy-preserving secret shared computations using mapreduce, IEEE Transactions on Dependable and Secure Computing 18 (2021) 1645–1666.

[41] H. Corrigan-Gibbs, D. Boneh, Prio: Private, robust, and scalable computation of aggregate statistics, in: 14th USENIX symposium on networked systems design and implementation, ACM Press, New York, NY, USA, 2017, pp. 259–282.

[42] P. Gupta, Y. Li, S. Mehrotra, N. Panwar, S. Sharma, S. Almanee, Obscure: Information-theoretically se-

cure, oblivious, and verifiable aggregation queries on secret-shared outsourced data, IEEE Transactions on Knowledge and Data Engineering 34 (2022) 843–864.

[43] F. Emekci, A. Methwally, D. Agrawal, A. ElAbbadi, Dividing secrets to secure data outsourcing, Information Sciences 263 (2014) 198–210.

[44] A. Shamir, How to share a secret, Communications of the ACM 22 (1979) 612–613.

[45] V. Attasena, J. Darmont, N. Harbi, Secret sharing for cloud data security: a survey, The VLDB Journal 26 (2017) 657–681.

[46] A. Chandramouli, A. Choudhury, A. Patra, A survey on perfectly-secure verifiable secret-sharing, ACM Computing Surveys 54 (2022) 1–36.

[47] C. Blundo, D. R. Stinson, Anonymous secret sharing schemes, Discrete Applied Mathematics 77 (1997) 13–28.

[48] S. Dolev, N. Gilboa, X. Li, Accumulating automata and cascaded equations automata for communicationless information theoretically secure multi-party computation, Theoretical Computer Science 795 (2019) 81–99.

[49] D. Servos, S. L. Osborn, Current research and open problems in attribute-based access control, ACM Computing Surveys 49 (2017) 1–45.

[50] L. Wang, D. Wijesekera, S. Jajodia, A logic-based framework for attribute based access control, in: Proceedings of the 2004 ACM workshop on formal methods in security engineering, ACM Press, New York, NY, USA, 2004, p. 45–55.

[51] K. Frikken, M. Atallah, J. Li, Attribute-based access control with hidden policies and hidden credentials, IEEE Transactions on Computers 55 (2006) 1259–1270.

[52] X. Zhang, Y. Li, D. Nalla, An attribute-based access matrix model, in: Proceedings of the 2005 ACM Symposium on Applied Computing, ACM Press, New York, NY, USA, 2005, p. 359–363.

[53] M. A. Hadavi, R. Jalili, E. Damiani, S. Cimato, Security and searchability in secret sharing-based data outsourcing, International Journal of Information Security 14 (2015) 513–529.

[54] W. Wang, Y. Hu, L. Chen, X. Huang, B. Sunar, Accelerating fully homomorphic encryption using gpu, in: Proceedings of 2012 IEEE Conference on High Performance Extreme Computing, IEEE Computer Society, Washington, DC, USA, 2012, pp. 1–5.

[55] A. A. Badawi, B. Veeravalli, J. Lin, N. Xiao, M. Kazuaki, A. K. M. Mi, Multi-gpu design and performance evaluation of homomorphic encryption on gpu clusters, IEEE Transactions on Parallel and Distributed Systems 32 (2021) 379–391.

[56] W. Wang, Z. Chen, X. Huang, Accelerating leveled fully homomorphic encryption using gpu, in: Proceedings of 2014 IEEE International Symposium on Circuits and Systems, IEEE Computer Society, Washington, DC, USA, 2014, pp. 2800–2803.

[57] Özgün Özerk, C. Elgezen, A. C. Mert, E. Öztürk, E. Savaş, Efficient number theoretic transform implementation on gpu for homomorphic encryption, The Journal of Supercomputing 78 (2022) 2840–2872.

[58] Y. Chen, Q. Zheng, Z. Yan, D. Liu, Qshield: Protecting outsourced cloud data queries with multi-user access control based on sgx, IEEE Transactions on Parallel and Distributed Systems 32 (2021) 485–499.

[59] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, M. Russinovich, Vc3: Trustworthy data analytics in the cloud using sgx, in: Proceedings of 2015 IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 2015, pp. 38–54.

[60] W. Sun, R. Zhang, W. Lou, Y. T. Hou, Rearguard: Secure keyword search using trusted hardware, in: Proceedings of IEEE Conference on Computer Communications, IEEE Computer Society, Washington, DC, USA, 2018, pp. 801–809.

[61] S. Bajaj, R. Sion, Trusteddb: A trusted hardware-based database with privacy and data confidentiality, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, ACM Press, New York, NY, USA, 2011, p. 205–216.

[62] J. Götzfried, M. Eckert, S. Schinzel, T. Müller, Cache attacks on intel sgx, in: Proceedings of the 10th European Workshop on Systems Security, ACM Press, New York, NY, USA, 2017, pp. 1–6.

[63] A. Moghimi, G. Irazoqui, T. Eisenbarth, Cachezoom: How sgx amplifies the power of cache attacks, in: Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems, Springer-Verlag, Berlin, Germany, 2017, p. 69–90.

[64] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, S. Mangard, Malware guard extension: abusing intel sgx to conceal cache attacks, Cybersecurity 3 (2020).

[65] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, A. Paverd, Mitigating branch-shadowing attacks on intel sgx using control flow randomization, in: Proceedings of the 3rd Workshop on System Software for Trusted Execution, ACM Press, New York, NY, USA, 2018, p. 42–47.

[66] K. Murdock, D. Oswald, F. D. Garcia, J. V. Bulck, D. Gruss, F. Piessens, Plundervolt: Software-based fault injection attacks against intel sgx, in: Proceedings of the IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 2020, pp. 1466–1482.

[67] S. Fei, Z. Yan, W. Ding, H. Xie, Security vulnerabilities of sgx and countermeasures: A survey, ACM Computing Surveys 54 (2022) 1–36.

[68] Y. Zhang, R. H. Deng, S. Xu, J. Sun, Q. Li, D. Zheng, Attribute-based encryption for cloud computing access control: A survey, ACM Computing Surveys 53 (2021) 83:1–41.

[69] P. P. Kumar, P. S. Kumar, P. J. A. Alphonse, Attribute based encryption in cloud computing: A survey, gap analysis, and future directions, Journal of Network and Computer Applications 108 (2018) 37–52.